

# Perfectly load-balanced, optimal, stable, parallel merge

Christian Siebert

RWTH Aachen University, Department of Computer Science  
 Laboratory for Parallel Programming  
 Schinkelstrasse 2a, 52062 Aachen, Germany  
[christian.siebert@rwth-aachen.de](mailto:christian.siebert@rwth-aachen.de)

Jesper Larsson Träff

Vienna University of Technology, Institute of Information Systems  
 Research Group Parallel Computing  
 Favoritenstrasse 16, 1040 Wien, Austria  
[traff@par.tuwien.ac.at](mailto:traff@par.tuwien.ac.at)

March 20, 2013

## Abstract

We present a simple, work-optimal and synchronization-free solution to the problem of stably merging in parallel two given, ordered arrays of  $m$  and  $n$  elements into an ordered array of  $m + n$  elements. The main contribution is a new, simple, fast and direct algorithm that determines, for any prefix of the stably merged output sequence, the exact prefixes of each of the two input sequences needed to produce this output prefix. More precisely, for any given index (rank) in the resulting, but not yet constructed output array representing an output prefix, the algorithm computes the indices (co-ranks) in each of the two input arrays representing the required input prefixes *without* having to merge the input arrays. The co-ranking algorithm takes  $\mathcal{O}(\log \min(m, n))$  time steps. The algorithm is used to devise a *perfectly load-balanced, stable*, parallel merge algorithm where each of  $p$  processing elements has *exactly* the same number of input elements to merge. Compared to other approaches to the parallel merge problem, our algorithm is considerably simpler and can be faster up to a factor of two. Compared to previous algorithms for solving the co-ranking problem, the algorithm given here is direct and maintains stability in the presence of repeated elements at no extra space or time cost. When the number of processing elements  $p$  does not exceed  $(m + n) / \log \min(m, n)$ , the parallel merge algorithm has optimal speedup. It is easy to implement on both shared and distributed memory parallel systems.

## 1 Introduction

We consider the problem of *stably* merging two ordered sequences in parallel. We assume the two sequences with  $m$  and  $n$  elements respectively to be stored in arrays. Elements have a key, and an ordering relation denoted by  $\leq$  is defined on the keys. The task is to produce an output sequence consisting of all input elements in order. Figure 1 illustrates the problem, where the height of each bar corresponds to the element key. Merging the two sequences sequentially can be done in  $\mathcal{O}(m + n)$  operations (cf. [10]). Most sequential algorithms are naturally stable, meaning that the relative order of elements with the same key is preserved. Figure 1 illustrates this: the equal-keyed elements  $\alpha$  and  $\beta$  both occur in the output with the element  $\alpha$  from the  $A$  array before the element  $\beta$  from the  $B$  array, and with all elements from  $A$  before  $\alpha$  also occurring before  $\alpha$  in the output array. Stability is important for many applications.

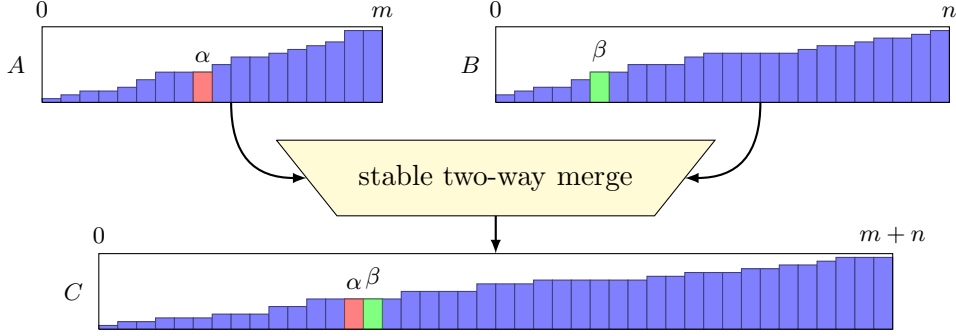


Figure 1: Merging takes two ordered input sequences stored in arrays  $A$  and  $B$  and produces an ordered output sequence stored in array  $C$ . Stability preserves the original order of equal-keyed elements, such as  $\alpha$  and  $\beta$ , with elements from  $A$  occurring before elements from  $B$ .

Many parallel two-way merge algorithms for large, ordered input arrays work as follows. In each of the arrays, a number proportional to the number of available processing elements of fixed, equidistant elements are chosen. For each the (*cross*)*rank*, which is the number of elements smaller than the chosen element in the other array, is determined via binary search. Cross ranks and indices of the chosen elements are used to determine disjoint array segments of the two input arrays that can be merged independently in parallel. On a parallel machine with  $p$  processing elements, the array segments to be merged by a single processing element can be guaranteed to be of size at most  $\lceil m/p \rceil$  and  $\lceil n/p \rceil$  respectively, but the size of the segments for different processing elements may differ by a factor of two. This pattern is found for instance in well-known PRAM (Parallel Random Access Machine, see, e.g., [8]) and BSP (Bulk Synchronous Parallel model, see, e.g., [15]) algorithms [5, 7, 9, 12]. Many of these algorithms use a separate merge step to determine the pairs of segments to merge. However, this extra step can be eliminated as shown in [14]. All of these parallel algorithms take  $\mathcal{O}((n+m)/p + \max(\log n, \log m))$  operations, are therefore work-optimal for  $p \leq (m+n)/\max(\log n, \log m)$ , such that the speedup compared to an optimal sequential algorithm is  $\mathcal{O}(p)$ . However, the load imbalance caused by the inexact determination of segments can limit the speedup to  $p/2$ . It is important to note that the dominant part of work in both parallel and sequential merge algorithm is done by the *same* (best) sequential merge algorithm.

In this paper, we show how the segments to merge can be determined such that all processing elements get *exactly* the same number of elements to merge, with a surprisingly simple and intuitive idea. For any index (*rank*) representing a prefix of the not yet computed, ordered output sequence the approach exactly determines the prefix (*co-rank*) of each of the input sequences that is needed to make up the given output prefix when the inputs are stably merged. To merge in parallel, the processing elements are simply assigned disjoint segments of the output (of roughly equal size) and they use the co-ranking algorithm to determine disjoint blocks of the two input arrays to merge. No synchronization is needed between the processing elements. For any given output rank  $i, 0 \leq i < m+n$ , co-ranks can be determined in  $\mathcal{O}(\log \min(m, n))$  operations and parallel merging can therefore be carried out in  $\mathcal{O}((m+n)/p + \log(\min(m, n)))$  operations per processing element.

This approach closes the gap up to a lower order term of  $\mathcal{O}(\log(\min(m, n)))$  between the actual number of operations to be carried out by parallel and sequential algorithms. Parallel algorithms that do the same number of operations with the same constant factors as the corresponding, best sequential algorithms are rare. A balanced number of operations is impor-

tant, because any additional constant factor overhead leads to a proportional loss of processing resources. In addition to these performance differences, the varying number of elements per processing element influences the memory consumption per processing element, which can be problematic especially for distributed-memory architectures.

Guaranteeing stability is sometimes problematic for parallel merge algorithms. Whenever stability is required by the application, a standard trick is to merge according to a lexicographic order on key-index pairs. From a practical point of view, this technique leads to undesirable extra compute and space costs. In the algorithm presented here, stability comes at no extra costs—neither additional index comparisons nor space consuming lexicographic orderings are required.

The co-ranking idea is not totally new. It was introduced in [1] and used subsequently in [3, 4] and [16, 17], but seems to have been somewhat overseen. In [3, 4] the co-ranking problem is solved only for the median, and the general problem reduced to the median case. The specific contribution in this paper is a simple, direct implementation that works for any output index and is stable by design. A distributed-memory implementation of a previous version is described in [13]. Some recent algorithms for merging achieve similarly good partitions [6, 11] and were in fact inspired directly by the algorithms in [3, 4]. We think the co-ranking algorithm presented next is more intuitive, with a simpler proof, and slightly better bounds.

## 2 A co-ranking algorithm

Let  $A$  and  $B$  be the two input arrays with  $m$  and  $n$  elements, respectively. We follow C programming language conventions and index arrays from 0. Both arrays are ordered according to an ordering relation  $\leq$  denoting comparison of element keys such that  $A[j-1] \leq A[j]$  for  $1 \leq j < m$ , and  $B[k-1] \leq B[k]$  for  $1 \leq k < n$ . Ultimately, we are interested in performing a stable merge of  $A$  and  $B$  into an array  $C$  with  $m+n$  elements. We denote this by  $C = \text{stable\_merge}(A, B, \leq)$ . For any  $i, 0 \leq i < m+n$  in  $C$  there is either a  $j, 0 \leq j < m$  such that  $C[i] = A[j]$  or a  $k, 0 \leq k < n$  such that  $C[i] = B[k]$ . Furthermore, for any  $i$ -element prefix  $C[0, \dots, i-1]$  of  $C$  there must be indices  $j$  and  $k$  of  $A$  and  $B$  such that  $C[0, \dots, i-1] = \text{stable\_merge}(A[0, \dots, j-1], B[0, \dots, k-1], \leq)$ . We will show in Lemma 1 that these  $j$  and  $k$  indices are indeed *unique*. They define the prefixes of  $A$  and  $B$  needed to form the prefix of  $C$  of length  $i$ . For an element  $C[i]$  we call the index  $i$  its *rank*, and the unique indices  $j$  and  $k$  its *co-ranks*. Consequently, we use the term *co-ranking* for the process of determining  $j$  and  $k$  from  $A, m, B, n$  and  $i$ . Figure 2 illustrates the co-rank definition and process.

Stability means that all equal elements of  $A$  should (in their relative order in  $A$ ) come before equal elements of  $B$  (also in their relative order in  $B$ ). That is, if  $C[i] = A[j]$  and  $A[j] = A[j+1]$  then also  $C[i+1] = A[j+1]$ , and if  $A[j] = B[k]$  then there is some  $i' > i$  for which  $C[i'] = B[k]$ . Stability is mostly easy to guarantee for sequential merging. We can therefore assume that we have an optimal sequential algorithm for stable merging at our disposal.

Let  $C[i-1]$  be the  $i^{\text{th}}$  element in the stably merged output array  $C$ . For determining the co-ranks  $j$  and  $k$  (and thereby determining whether  $C[i-1]$  comes from the  $A$  or the  $B$  array), we first note that  $j+k=i$ . The  $i^{\text{th}}$  output element  $C[i-1]$  is either  $A[j-1]$  or  $B[k-1]$ , both elements  $A[j-1]$  and  $B[k-1]$  are in the prefix  $C[0, \dots, i-1]$ , but neither of  $A[j]$  nor  $B[k]$  are. (For convenience we assume that  $A[-1] = -\infty, A[m] = \infty$ , and likewise for  $B$ . However, these sentinels do not have to be stored.) If  $C[i-1] = A[j-1]$  (that is, the  $i^{\text{th}}$  output element comes from  $A$ ), then it must hold that  $A[j-1] \leq B[k]$ . If instead  $C[i-1] = B[k-1]$ , then likewise  $B[k-1] \leq A[j]$ . Now, since the merge is stable, it cannot be that  $B[k-1] = A[j]$  since that would mean that an element of  $B$  equal to an element of  $A$  comes before the  $A$  element in the output array. Therefore, in this case  $B[k-1] < A[j]$ .

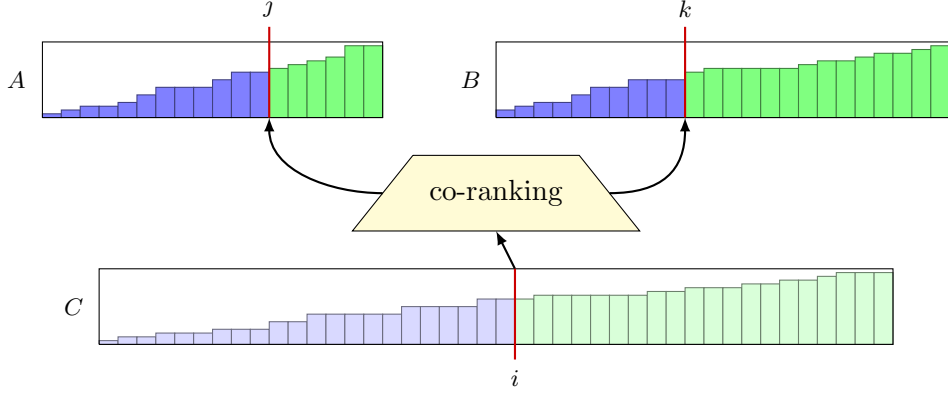


Figure 2: Co-ranking determines for any given rank (index)  $i$  in  $C$  the co-ranks  $j$  and  $k$  in  $A$  and  $B$  without having to actually merge  $A$  and  $B$ .

**Lemma 1** *For any  $i, 0 \leq i < m + n$ , there exists a unique  $j, 0 \leq j \leq m$ , and a unique  $k, 0 \leq k \leq n$ , with  $j + k = i$  such that*

1.  $j = 0 \vee A[j - 1] \leq B[k]$  and
2.  $k = 0 \vee B[k - 1] < A[j]$ .

*These  $j$  and  $k$  fulfill  $\text{stable\_merge}(A[0, \dots, j - 1], B[0, \dots, k - 1], \leq) = C[0, \dots, i - 1]$  where  $C = \text{stable\_merge}(A, B, \leq)$ .*

We will refer to (1) and (2) as the first and the second Lemma condition, respectively.

*Proof:* We need to argue for the uniqueness of  $j$  and  $k$ . Assume the two conditions hold for some  $0 \leq j < m$  with a corresponding  $k$  such that  $j + k = i$ . We look at  $j + 1$  and  $j - 1$ : for  $j + 1$ ,  $A[j] \leq B[k - 1]$  contradicts  $B[k - 1] < A[j]$ , and for  $j - 1$ ,  $B[k] < A[j - 1]$  contradicts  $A[j - 1] \leq B[k]$ . Thus neither  $j - 1$  nor  $j + 1$  can fulfill both conditions, and since the arrays are ordered, neither can any other smaller or larger indices. For the existence we argue as follows. Let  $j$  with  $0 < j < m$  be the highest indexed element such that  $A[j - 1] \leq B[k]$  with a corresponding  $k, 0 < k < n$ ; since  $A[j] > B[k - 1]$  the second Lemma condition is also fulfilled. If no such element exists, we check for  $j = 0$  and  $k = 0$ . For  $j = 0$  the first lemma condition is trivially true, and the second lemma condition  $B[k - 1] < A[0]$  holds if the length  $k$  prefix of  $B$  comes before  $A$  in the output array. For  $k = 0$  the second lemma condition is trivially true, and the first lemma condition  $A[j - 1] \leq B[0]$  holds if the length  $j$  prefix of  $A$  comes before  $B$  in the output array. Since either must be true, indices  $j$  and  $k$  with  $j + k = i$  must exist as claimed.  $\square$

Lemma 1 immediately gives an approach to find the co-ranks  $j$  and  $k$  for any  $i$  efficiently. Algorithm 1 implements this approach and finds the unique  $j$  and  $k$  fulfilling both conditions of Lemma 1. It maintains the invariant  $j + k = i$ , and works similar to a binary search. The **and** in lines 6 and 11 denotes a “conditional and” as in the C language and means that the condition at the right is only evaluated if the condition on the left evaluates to true. The algorithm starts with the extreme assumption that all  $i$  elements come from  $A$  (as far as possible), that is setting  $j = \min(i, m)$  and  $k = i - j$ . For both arrays  $A$  and  $B$  it maintains lower bounds  $j_{\text{low}}, j_{\text{low}} \leq j$ , and  $k_{\text{low}}, k_{\text{low}} \leq k$  on the prefix lengths. Furthermore, it maintains the invariant that either the unique index in array  $A$  fulfilling the conditions of Lemma 1 is between  $j_{\text{low}}$  and  $j$  or the unique index in array  $B$  fulfilling the condition is between  $k_{\text{low}}$  and  $k$ .

---

**Algorithm 1**  $\text{co\_rank}(i, A, m, B, n, \leq)$  determines for any given  $i, 0 \leq i < m + n$  the unique co-ranks  $j$  and  $k$  in arrays  $A$  and  $B$ .

---

```

1:  $j \leftarrow \min(i, m)$ 
2:  $k \leftarrow i - j$  {Invariant:  $j + k = i$ }
3:  $j_{\text{low}} \leftarrow \max(0, i - n)$ 
4:  $\text{active} \leftarrow \text{true}$ 
5: while  $\text{active}$  do
6:   if  $j > 0 \wedge k < n$  and  $A[j - 1] > B[k]$  then
7:     {First Lemma condition violated: decrease  $j$ }
8:      $\delta \leftarrow \lceil \frac{j - j_{\text{low}}}{2} \rceil$ 
9:      $k_{\text{low}} \leftarrow k$ 
10:     $j, k \leftarrow j - \delta, k + \delta$ 
11:   else if  $k > 0 \wedge j < m$  and  $B[k - 1] \geq A[j]$  then
12:     {Second Lemma condition violated: decrease  $k$ }
13:      $\delta \leftarrow \lceil \frac{k - k_{\text{low}}}{2} \rceil$ 
14:      $j_{\text{low}} \leftarrow j$ 
15:      $j, k \leftarrow j + \delta, k - \delta$ 
16:   else
17:     {No conditions violated: unique  $(j, k)$  found}
18:      $\text{active} \leftarrow \text{false}$ 
19:   end if
20: end while
21: return  $(j, k)$ 

```

---

If the first Lemma condition is violated, that is  $A[j - 1] > B[k]$ , then  $j$  is too large and therefore must be decreased. This is done by cutting the size of the interval from  $j_{\text{low}}$  to  $j$  in half. In order to maintain the invariant  $j + k = i$ ,  $k$  is correspondingly increased at the same time. In order to prevent that  $k$  exceeds  $n$ , the lower bound  $j_{\text{low}}$  is initially set to  $\max(0, i - n)$ . When  $j$  is decreased, the lower bound  $k_{\text{low}}$  can be increased to  $k$  because no smaller  $k$  could fulfill the first Lemma condition (that would mean a larger  $j$ , which cannot be). If instead the second Lemma condition is violated, that is  $B[k - 1] \geq A[j]$ , then  $k$  needs to be decreased. Again, we do this by halving the size of the interval from  $k_{\text{low}}$  to  $k$ . Since no smaller  $j$  can now fulfill the first Lemma condition, the lower index  $j_{\text{low}}$  can be increased to  $j$ . The algorithm terminates when both conditions are fulfilled, which will happen at the latest when either  $j - j_{\text{low}} = 0$  or  $k - k_{\text{low}} = 0$ . Note that in the first iteration, only the first Lemma condition can be violated, as per initialization either  $j = m$  or  $k = 0$ . The lower bound  $k_{\text{low}}$  will therefore be set in the first iteration. A first few possible iterations of the algorithm are illustrated in Figure 3.

**Proposition 1** *Algorithm 1 computes the co-ranks  $j$  and  $k$  for ordered arrays  $A$  and  $B$  with  $m$  and  $n$  elements for any  $0 \leq i \leq m + n$ . The algorithm requires at most  $\lceil \log_2 \min(m, n, i, m + n - i) \rceil$  iterations and thus comparisons of element keys.*

*Proof:* The algorithm clearly maintains the invariant  $j + k = i$ . We claim further that either the co-rank for array  $A$  lies between  $j_{\text{low}}$  and  $j$  or the co-rank for array  $B$  lies between  $k_{\text{low}}$  and  $k$ . This holds before the first iteration, as  $j_{\text{low}}$  and  $j$  are set assuming that as many of the  $i$  output elements as possible come from  $A$  (see Figure 3). Assume that the invariant holds before an iteration starts. If the condition in Line 6 holds, then the correct index in array  $A$  must be between  $j_{\text{low}}$  and  $j$ . For the next iteration the correct index in  $A$  is either between  $j_{\text{low}}$  and  $j - \delta$  or between  $j - \delta$  and  $j$ . In the latter case, the co-rank for array  $B$  must be between  $k$

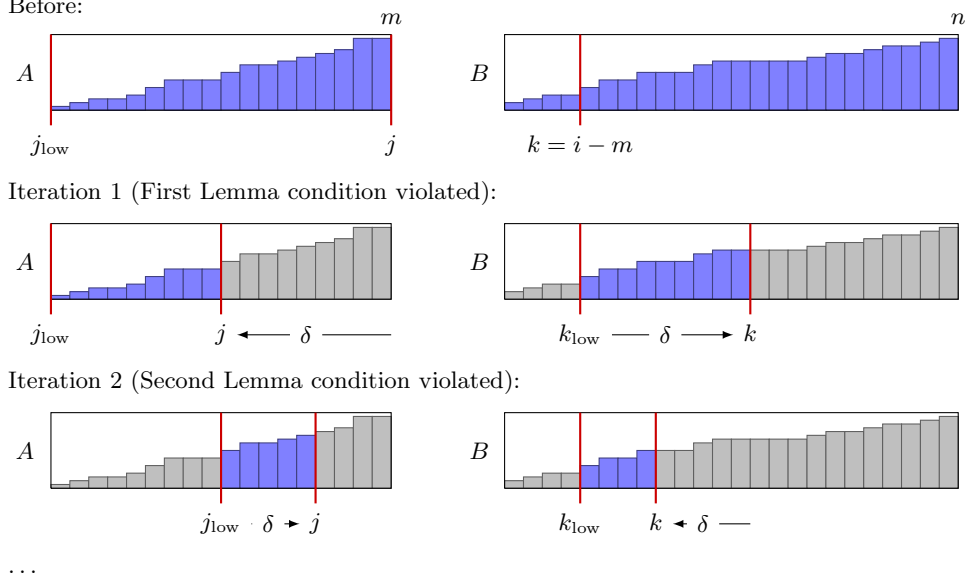


Figure 3: A first few possible iterations of the co-ranking algorithm. In the first iteration only the first condition can be true, so the interval from  $j_{\text{low}}$  to  $j$  is halved. The lower bound  $k_{\text{low}}$  is set accordingly, and  $k$  is increased by the same  $\delta$ . In the second iteration, assuming the second condition to be true, instead the interval from  $k_{\text{low}}$  to  $k$  is halved.

and  $k + \delta$ , and the invariant is maintained by setting the lower bound  $k_{\text{low}}$  to  $k$  and increasing  $k$  by  $\delta$ . Similarly, if instead the condition in line 11 evaluates to true, the correct index in  $B$  must be between  $k_{\text{low}}$  and  $k$ , and dividing the interval and increasing the lower bound  $j_{\text{low}}$  for array  $A$  likewise reestablishes the invariant. It therefore holds also after the iteration. After the first iteration,  $k - k_{\text{low}} = \delta$  (if the first condition is true, otherwise the algorithm terminates) and  $j - j_{\text{low}} \leq \delta$ . As  $j$  is initialized to  $\min(i, m)$  and  $j_{\text{low}} \geq 0$ , trivially  $j - j_{\text{low}} \leq \min(i, m)$ . In order for  $k$  not to exceed  $n$ ,  $j_{\text{low}}$  must be initialized such that  $k + (j - j_{\text{low}}) < n$ , that is  $i - j_{\text{low}} < n$ ;  $j_{\text{low}}$  is therefore set to  $\max(0, i - n)$ . Assume  $j_{\text{low}} = i - n \geq 0$ ; since  $j$  is initialized to either  $m$  or  $i < m$ , it follows that either  $j - j_{\text{low}} \leq m + n - i$  or  $j - j_{\text{low}} \leq n$ . As  $\delta$  is halved in each subsequent iteration and  $j - j_{\text{low}}$  starts out being  $\min(m, n, i, m + n - i)$ , at most  $\lceil \log_2 \min(m, n, i, m + n - i) \rceil$  iterations are required.  $\square$

### 3 Parallel merging

The co-ranking algorithm provides a simple and efficient way of performing merging in parallel. Let  $p$  processing elements be given, all of which can access input and output arrays  $A$ ,  $B$  and  $C$ . Each processing element has an own id  $r$ ,  $0 \leq r < p$ . Each processing element independently computes the start and end indices of a block of the output array from  $C[i_r, \dots, i_{r+1} - 1]$ . The output blocks can be chosen such that they partition the whole output array, and differ in size by at most one element. Each processing element computes for both start and end index the corresponding co-ranks. These co-ranks determine the (disjoint) blocks of the input arrays this processor need to merge sequentially to compute its output block. This is shown in detail as Algorithm 2.

**Proposition 2** *Algorithm 2 merges ordered arrays  $A$  and  $B$  of  $m$  and  $n$  elements stably using*

---

**Algorithm 2** Synchronization-free parallel merging of ordered arrays  $A$  and  $B$  for processing element  $r$ ,  $0 \leq r < p$

---

- 1:  $i_r \leftarrow \lfloor r \frac{m+n}{p} \rfloor$  {Start index of output block}
  - 2:  $i_{r+1} \leftarrow \lfloor (r+1) \frac{m+n}{p} \rfloor$  {End index of output block}  
 {To avoid synchronization processing element  $r$  computes co-ranks for both start and end index}
  - 3:  $(j_r, k_r) \leftarrow \text{co\_rank}(i_r, A, m, B, n, \leq)$
  - 4:  $(j_{r+1}, k_{r+1}) \leftarrow \text{co\_rank}(i_{r+1}, A, m, B, n, \leq)$
  - 5:  $\text{stable\_merge}(A[j_r, \dots, j_{r+1} - 1], B[k_r, \dots, k_{r+1} - 1], C[i_r, \dots, i_{r+1} - 1], \leq)$
- 

$p$  processing elements. The number of elements to merge per processing element is at most  $\lceil \frac{m+n}{p} \rceil$ , and the total time complexity is  $\mathcal{O}(\frac{n+m}{p} + \log \min(m, n))$ .

Algorithm 2 avoids synchronization by letting each processor compute the co-ranks for both start and end index. If synchronization is inexpensive, half the co-ranking work can be saved by letting each processing element read the co-ranks for its end index from the next processing element. A synchronization step is in this case required after the co-ranks computation. It is also worth noting that in the co-ranking procedure, when invoked in parallel by several processing elements, concurrent reading of locations may easily occur. However, it may well be possible to eliminate these by a careful pipelining such as in [2, 7]. The implementation given here will run efficiently on a CREW PRAM.

Stability follows from the properties of the co-ranking procedure, and the use of a stable, sequential merge algorithm. The number of elements to merge per processing element differs at most by one element. These are the main improvements over previous parallel merge algorithms, where the number of elements to merge, although  $\mathcal{O}((n+m)/p)$ , can differ by a factor of two.

Algorithm 2 assumes a shared-memory parallel system. The algorithm can, however, easily be adapted to distributed memory systems as shown in [14]. This is a considerably more elegant, better, and easier implementation than for instance the BSP algorithm presented in [5].

## References

- [1] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1369, 1987.
- [2] D. Z. Chen. Efficient parallel binary search on sorted arrays, with applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):440–445, 1995.
- [3] N. Deo, A. Jain, and M. Medidi. An optimal parallel algorithm for merging using multis-election. *Information Processing Letters*, 50(2):81–87, 1994.
- [4] N. Deo and D. Sarkar. Parallel algorithms for merging and sorting. *Information Sciences*, 56(1–3):151–161, 1991.
- [5] A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27(6):809–822, 2001.
- [6] O. Green, R. McColl, and D. A. Bader. GPU merge path: a GPU merging algorithm. In *ACM International Conference on Supercomputing (ICS)*, pages 331–340, 2012.
- [7] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

- [8] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [9] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. *Informatique Théoretique et Applications*, 27(4):295–310, 1993.
- [10] D. E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.
- [11] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *Workshop on Multi-Threaded Architectures and Applications (MTAAP), 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1611–1618, 2012.
- [12] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.
- [13] C. Siebert and J. L. Träff. Efficient MPI implementation of a parallel, stable merge algorithm. In *Recent Advances in Message Passing Interface. 19th European MPI Users’ Group Meeting*, volume 7490 of *Lecture Notes in Computer Science*, pages 204–213. Springer, 2012.
- [14] J. L. Träff. Simplified, stable parallel merging. CoRR abs/1202.6575, 2012.
- [15] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [16] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: algorithm and implementation results. *Parallel Computing*, 15(1-3):165–177, 1990.
- [17] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:171–177, 1991.